CVS $Revision: 1.11 $Date: 2003/11/03 23:37:07

Physics Simulation Design

G. Bernstein, with Kim/Kushner/Kuznetsova/Gladney/Tucker/Lamoreaux

1. Purpose of the Document

This document will eventually specify all the classes (i.e. Nouns and Verbs) that are needed to implement the desired SNAP simulations. Initially the development will focus on the SNIa investigations, but the class structure will be designed to accomodate future weak lensing and other use cases.

The prerequisites are: the *Simulation Architecture* document (GK), which specifies the system protocols for SNAPSim; and the data flow diagrams (AK) that outline the primary uses for SNAPSim. The old "block diagram"(GB) might closely resemble what is produced here.

In these drafts, **???** mark places that are in need of further work or the opinions of the design group. Objectst that will correspond to a Java class are written in `teletype font.`

2. Domains

It will be useful to divide the physics simulation into different *domains.* These domains do not (yet) correspond to any formal programming structure; they are meant as a way of organizing the problem into nearly separable concepts/tasks, hence will also help in dividing the labor of design/implementation. The guidelines for the definition of domains are:

1. Nouns, or data structures in general, will belong to only one domain. The exceptions are mathematical utility classes.
2. Verbs, or algorithms in general, will sometimes cross domains, *e.g.* by producing Nouns in a new domain from data in previous domains, but the number of involved domains will be kept to a minimum.
3. The branch points for simulations or use cases will tend to be after the completion of a domain. For example, when testing the effect of an instrument specification change, the Universe and Mission domains will be held fixed while we loop over changes to Observatory domain. Tests of different analysis algorithms will start with common SourceData and loop over various Analysis cases.

Following sections define the domains. The data members within each domain are listed in later sections.

Utility: Mathematical/physical classes that are used throughout many domains, such as `WavelengthFunction`, `SphericalCoordinate` or `PSF`.

Universe: everything that happens before the photons hit the telescope or the top of the atmosphere, including `AstronomicalSources`, their distributions in the sky, and intervening effects such as dust and lensing.

Observatory: description of the hardware that converts photons to image data, from the top of the atmosphere to the detector output.

Mission: List of the pointings, times, and configuration of the observatory during exposures, and the algorithm for determining this schedule.

PixelData: the simulated (or real) bits that flow from the spacecraft, along with pixel-level calibrations such as bias and flat-field.

SourceData: the quantities of interest for sources that are measured directly from the images. For points sources, there is `PointSourceDatum` that specifies the position, flux, and uncertainties in these quantities in a single exposure. Spectra and light curves are arrays of `PointSourceDatum` objects. For extended sources, there are additional observables related to shape. There should be no cross-correlation of random errors between different SourceData members, as these are the most basic observable quantities of the objects.[1] SourceData will be in nominally calibrated physical units, discussed further in §10.

Calibration: classes that embody the model used to convert the nominal fluxes in the SourceData into absolute normalizations of the spectra of sources with some assumed spectrum. A simple case is to assign a single photometric zeropoint to each `Channel` (*cf.* §7), but more complex models involve color terms, etc. In other words the Calibration domain takes SourceData for a chosen source and produces an estimate of its absolute flux normalization. There will also astrometric Calibration classes.

Analysis: results of subsequent processing of calibrated SourceData, for instance light curve or spectral parametric fits, fitted supernova models, cosmology fits, lensing maps, etc. Will usually consist of probability distributions over some space of derived quantities, *i.e.* fitted model parameters.

These domains are ordered, in that creation of a Noun in one domain should require only information from previous domains. The primary exception that I can think of is that the Mission scheduling algorithm will end up being forward-looking once we are using partially-analyzed SN data as a spectroscopy trigger (or any other adaptive scheduling technique). Another small "backflow" of information is that when we realize a Universe, we don't want to do the entire sky, just the parts

---

[1] In fact there might be cross-correlations between properties of objects that overlap on the sky.

that are likely to be observed by a given Mission. So some Mission information must be provided to build the Universe.

3. General Considerations

There are a few design considerations that are applicable to many or all of the domains.

3.1. Truth, Realization, and Model

A full simulation needs to contain multiple code objects to represent the same physical object. There is typically some *truth* version of the object, representing the behavior that is assumed in constructing the artificial measurements. Then there is a *model* version of the object, which is the best estimate of the object's behavior resulting from analysis of the measurements.

Likewise for measurements, there is typically a true value of the expected number of counts (or flux, or position, etc.) and an expected variance—or, more generally, a probability distribution of the measurement. Then there is a *realization* of the measurement, which is a single value drawn from the probability distribution. Equivalently this is the best estimate of the true value of the flux, given the measurement. There is usually an estimate of the variance, which differs from the true variance.

A general goal of our simulation should be that **truth and model of an object, or truth and realization of a measurement, should be embodied by code that is as similar as possible, preferably by the same code.** The interface for `SupernovaIa`, or in fact any `AstronomicalSource` or effect, should be capable of serving the needs of data *simulation* (truth) as well as data *analysis* (model). In a given simulation run, we might analyse the data with a different implementation (*i.e.* model) of `SupernovaIa` for analysis than we used for simulation. But each implementation should be capable of doing either job.

This is important for several reasons:

- In general, a given bit of information, such as a source model, should be coded only once, in order to avoid having two versions that can get out of sync during code development.
- The analysis code development always includes a test that it can properly recover the truth parameters. This is facilitated by having the exact truth model be used by the analysis.
- We want to smooth the transition from simulation pipeline to reduction pipeline; in the latter, there is no known truth.
- Both the simulation and the analysis are essentially the process of modelling the observations—one produces observations from source info, the other vice-versa—so they logically should

share the same code.

Perhaps the only significant consequence of this guideline is that models should be capable of providing the derivatives of observables with respect to model parameters. Such derivatives greatly speed the analysis phase.

Another guideline in this area is that **truth and model (or truth and realization) should be stored in distinct instances of the model class.** Any correspondence between truth and model instances should be recorded in an external catalog class, not within either the truth or the model class. There will be many simulation runs in which there is, for example, truth data but no realization (*e.g.* in a Fisher analysis), and many pixel-based simulation runs in which there are model objects that have no corresponding truth (*e.g.* false positives or a blind analysis). Of course the latter is true for the real data as well!

3.2. Fundamental Operations

A SNAPSim task typically involves four general phases:

1. Specify the Universe being observed, the Observatory being used, and the Mission strategy, then realize each of these.
2. Determine which targets of interest were observed at what times with what hardware channels.
3. Realize the SourceData for the each observation of each target (with or without realizing PixelData).
4. Analyze (and perhaps calibrate) the SourceData to obtain final quantities of interest, typically the uncertainty on dark energy parameters.

A goal of this design is to make each of the first three processes as generic as possible, in the sense that our data structures can flexibly handle different kinds of observatories, astronomical objects, spectral vs imaging data, etc., without having to recode the Verbs that do these steps. This means using polymorphism and interfaces wisely.

The following subsections list some operations that seem very basic to the simulation process, so many of the interfaces will be designed to work smoothly with these Verbs/facilities. **??? Are these going to be verbs?**

3.2.1. Astrometric Calculator

In Step 2, it is necessary to determine which sources are projected within the borders of a given detector in the focal plane. This requires some efficient means of calculating the sky-coordinate boundaries of the detector given the optical distortions, the pointing and orientation of the telescope, and distortions induced by atmospheric and gravitational lensing. We need to develop an Astrometric Calculator which can implement these calculations efficiently for a wide variety of distortion effects that are distributed between the Universe, Observatory, and Mission domains. The Astrometric Calculator will be embodied by the coordinate and distortion mapping classes of §coordinates.

3.2.2. Exposure Time Calculator

Step 3, realization of SourceData, is a fundamental operation, and is essentially the job of the Exposure Time Calculator (ETC) when we are not simulating PixelData. For Fisher analyses, we will ask the ETC only to calculate the variance of the measurement, and we do not need the realization itself. The ETC is another facility that needs to be efficient and flexible.

The ETC, to realize a point-source measurement, needs to know from different domains:

- From the Mission:
  - exposure time
  - exposure repetition pattern (dithering, etc)
- From the Observatory:
  - pixel pitch and plate scale
  - detector noise model
  - cosmic ray model
- From combination of Universe and Observatory:
  - target count rate (integrated over band)
  - background count rate (integrated over band)
  - ePSF (effective PSF, includes all optical & detector contributions)

For extended-source analysis such as galaxy photometry or shape analysis, also needs to know a galaxy `Shape`, from the Universe domain.

So these are all quantities that we should be prepared to derive for any type of astronomical target and any type of observatory setup. The `Observation` class will hold all this information (§5).

3.2.3. Image Simulator

The exact same information that the ETC needs is also required to create pixel-level simulations. A fast and flexible "draw" routine will be needed, which can execute distortions and convolutions of the intrinisic source shapes.

3.2.4. Model Fitting

Most Analysis activities, and many Calibration activities, will involve fitting a parametric model of the sources, dust, cosmology, and instrument calibration to a series of SourceData or higher-level data. **???** We need to work out a syntax for flexibly specifying parameter sets that consist of elements of various Nouns, for specifying figure-of-merit functions for optimizing, and for holding covariance matrices or likelihood functions of resultant fits.

4. Utility Classes

4.1. Spectral Information

We will be awash in functions of wavelength: source spectra, extinctions, background brightnesses, QE's, etc., that have to be multiplied and integrated. A uniform interface is needed:

```
interface WavelengthFunction {
  double          valueAt(double lambda);
  void            validity(double& lambdaMin, double& lambdaMax);
  void            support(double& lambdaMin, double& lambdaMax);
  SpectralUnits   units();
  double          resolution();
  ???             preferredValues();
}
```

Each implementation embodies some function $f(\lambda)$, which is defined in the region of `validity` $[\lambda_{\rm min}, \lambda_{\rm max}]$. Requesting the `valueAt` a $\lambda \notin [\lambda_{\rm min}, \lambda_{\rm max}]$ will throw an exception. **??? define exception class.**

$f(\lambda)$ is gauranteed to be zero outside the interval of `support`. Filter functions will have the support region contained within the definition region, but source spectra may be non-zero outside the bounds of their definition.

The `resolution` and `preferredValues` are hints to the `SpectrumIntegrator`: the former suggests the largest allowable $\ln\lambda$ step that will capture the variation of the function, and the latter may contain information about the nature of a tabulated function.

4.1.1. Units

The functions carry a tag to indicate their units. This is a Noun that we design and serves as a type-safety mechanism, for instance insuring that a QE is not used where a source spectrum is required.

```
class SpectralUnits extends Noun {
private:
???
public:
     bool equals(SpectralUnits rhs) const;
     SpectralUnits times(SpectralUnits rhs) const;
     static final SpectralUnits Luminosity;
     static final SpectralUnits Flux;
     static final SpectralUnits Area;
     static final SpectralUnits SolidAngle;
     static final SpectralUnits SurfaceBrightness;
     static final SpectralUnits TransferFunction;
     static final SpectralUnits Rate;
}
```

The units of these quantities should be standardized throughout the simulation. The most robust units for luminosities are photons per ln(wavelength) per second—luminosities expressed this way are independent of energy or wavelength units; the value is the same as per ln(frequency); evolve only as $(1+z)$ under cosmological redshift because the photon count and logarithmic intervals are invariants; and convert naturally to electrons in the detectors. If we adopt μm as the wavelength unit, m^2 as the collecting-area unit, and sr as the solid-angle unit, then the SpectralUnits constants are then as given in Table 1. The `TransferFunction` units are appropriate for extinction, atmospheric/atmospheric/filter transmission curves, and QE curves.

??? Do we want to distinguish "specific" quantities (per log wavelength) from the integrated quantities?

4.1.2. Implementations of `SpectralFunction`

We will have numerous implementation of the `SpectralFunction` interface. These will certainly include

```
class ScalarSpectralFunction extends Noun implements SpectralFunction;
```

```
class TabularSpectralFunction extends Noun implements SpectralFunction;
```

```
class BoxcarSpectralFunction extends Noun implements SpectralFunction;

class ProductSpectralFunction extends Noun implements SpectralFunction {
private:
    NounArray factors;
public:
    ????
}
```

The `ProductSpectralFunction` is defined to be the product of all its factors, each of which is a `SpectralFunction`. For the `ProductSpectralFunction`, the `support` and `validity` ranges are the intersections of all its factors'. The resolution is the minimum of its factors'. The `factors` array will hold **??? links to** the component functions. There will be methods for adding factors, etc.

4.1.3. The Integrator

A final class in this set is the utility to integrate a `SpectralFunction`. This can be a Verb **???**.

```
class SpectrumIntegrator implements Verb {
    <required Verb stuff...>
    double integrate(SpectralFunction f, SpectralUnits units);
}
```

The `SpectrumIntegrator` returns

$$\int_{\lambda_{\min}}^{\lambda_{\max}} f(\lambda)\, d\ln\lambda \tag{1}$$

with the integral over the `support` of the `SpectralFunction` f. An exception is thrown if the `validity` does not span the `support`. The units of the integral quantity are output. They will tend to be `SpectralUnits::Rate`.

The `SpectrumIntegrator` should be intelligent enough to use `f.resolution()` and to extract the scalar quantities from a `ProductSpectralFunction` before integration.

It will be desirable to have `SpectrumIntegrator` cache the results of its integrations, because the same ones will likely be repeated many times, *e.g.* the flux of a supernova at peak integrated over the B band.

4.2. Astrometric Coordinates and Maps

Positions on the sky will need to be specified by an abstract class

```
class SphericalCoordinates extends Noun {
private:
    double x,y,z;   //Three direction cosines
    abstract void convertToICRS(double x, double y, double z);
    abstract void convertFromICRS(double x, double y, double z);
public:
    ????
}
```

Derived classes are then constructed for each specific coordinate system:

```
class SphericalCoordinatesICRS extends SphericalCoordinates;
class SphericalCoordinatesEcliptic extends SphericalCoordinates;
class SphericalCoordinatesGalactic extends SphericalCoordinates;
class TangentPlaneCoordinates extends SphericalCoordinates;
```

The last of these specifies spherical coordinates as a tangent-plane projection about a chosen axis on the sky. It requires a class

```
class Orientation extends Noun {
    SphericalCoordinates axis;
    double  positionAngle;
}
```

We will also eventually require a class to specify locations in 3d space:

```
class CartesianCoordinates extends Noun;
class CartesianICRS extends CartesianCoordinates;
class CartesianIGRS extends CartesianCoordinates;
```

The first derived class gives coordinates relative to the Solar System barycenter, in the ICRS orientation frame. The second gives coordinates relative to the geocenter in coordinates affixed to rotating Earth. They are needed for keeping track of the observatory location, essential for such tasks as planetary ephemerides, parallax calculations, and observing circumstances.

Coordinate mappings will arise in both the Universe and the Observatory domains, and will be modelled or used in the Calibration and Analysis domains. There will be a coordinate-transform class:

```
interface Distortion {
  void forward(SphericalCoordinates xyTrue, SphericalCoordinates& xyObs)
  void inverse(SphericalCoordinates xyObs,  SphericalCoordinates& xyTrue)
  Matrix magnification(SphericalCoordinates xyObs);
  double fluxMagnification(SphericalCoordinates xyObs);
}
```

The `magnification()` method gives the local magnification matrix, *i.e.* the linear expansion of the distortion map about a given line of sight, and `fluxMagnification` returns the Jacobian of this map.

The implementations of `Distortion` will include some gravitational lensing cases [**???** lensing distortion depends upon the redshifts of the source and lens; we need some way to incorporate this] from the Universe; a map of the optical distortions of the telescope about the optic axis; and perhaps additional classes

```
class StellarAberration extends Noun implements Distortion {
private:
   CartesianCoordinates velocity
...
}
class AtmosphericRefraction extends Noun implements Distortion {
}
```

There is an implementation that is the composition of several `Distortions`:

```
class DistortionProduct extends Noun implements Distortion;
```

unlike `PSF` and `SpectralFunction` compositions, the order of `Distortions` is significant.

??? The domain of `Distortion` functions needs a little work. For a given Observatory `Channel` we will want a `Distortion` that is actually a map from pixel coordinates on the focal plane (in μm probably) into a `TangentPlaneCoordinate` which is in radians but is relative to the optic axis. The `Orientation` of the optic axis then turns this into a map into `SphericalCoordinatesICRS`.

A region on the sky can be described through this interface:

```
interface SolidAngle {
  bool includes(SphericalCoordinates point);  //true if region includes the point
  bool overlaps(SolidAngle rhs);  //true if region touches another region
  bool includes(SolidAngle rhs);  //true if region includes another region
  double area();              // returns steradians enclosed
}
```

Instead of being an interface with implementations of many kinds of regions, we might just restrict ourselves to polygonal regions. Otherwise the coding of `overlaps` and `includes` could be messy.

4.3. Time

A class for a UT date and time:

```
class UT extends Noun {
  double interval(UT rhs);   //time interval (s) between this and rhs
  bool   precedes(UT rhs);
  UT     plus(double seconds);
}
```

4.4. Point Spread Functions

The blurring of the astronomical image before and during detection is described by a point spread function (PSF):

```
interface PSF {
  complex kValue(double kx, double ky);
  complex xValue(double x, double y);
  double  maxK();          //largest significant k-vector
  double  stepK();                     //suggested k-space resolution
  bool    isAxisymmetric();
  // Other calls that specify the spatial extent, k-space extent,
  // and various moments.
}
```

We can expect to require several implementations

```
class PSFGaussian extends PSF;
class PSFAiry extends PSF;
class PSFKolmogorov extends PSF;
class PSFBox extends PSF;
class PSFTabularK extends PSF;
class PSFProduct extends PSF;
}
```

The Gaussian is a useful analytic PSF, and accurately describes the charge diffusion contribution. The Kolmogorov form arises from atmospheric seeing; the Box is the typical effect of pixelization;

`PSFTabularK` allows a PSF that is tabulated in k space.

The last implementation of this interface is a PSF that consists of a product of other PSFs in a NounArray. It can propagate `maxK()`, `stepK()` and `isAxisymmetric()` in the expected fashion. A clever version would be capable of recognizing efficent analytic combinations, for example all `PSFGaussian` contributions can be combined into a single one.

??? The units of PSFs need to be specified. Normally the domain of a PSF measured in arcseconds or radians. For detector effects, however, the domain is sometimes specified in microns. Perhaps we can standardize on radians, and leave it to the `Detector` classes to take a plate scale and provide a `PSF` that is in radian units. The output units of a PSF are usually such that the integral over its domain is unity (value at $k = 0$ is unity). A pixel response function (PRF), however, is a PSF that specifies the conversion from incident intensity ($\gamma\,\mathrm{s}^{-1}\,\mathrm{sr}^{-1}$) to the count rate in a pixel ($\gamma\,\mathrm{s}^{-1}$), so that its integral must have units of sr.

4.5. Parameter Sets and Model Fitting

Fitting of a model to a set of observations is going to be an oft-used process. A standardized representation of the parameters, data, likelihood functions, and uncertainties should be implemented.

The general fitting problem can be posed this way: we have a vector of observed quantities $\mathbf{o} = \{o_1, o_2, \ldots\}$, and a model might predict values $\hat{\mathbf{o}}$ for these observables. Most generically there is a (negative log) likelihood function $\mathcal{L}(\mathbf{o}|\hat{\mathbf{o}})$ over these observables. In the simplest case of independent Gaussian observables, this is the χ^2 function

$$\mathcal{L} = \sum_i \frac{(o_i - \hat{o}_i)^2}{\sigma_i^2}, \tag{2}$$

where σ_i^2 are the variances.

A model takes a vector of parameters $\mathbf{p}$ and produces a resultant $\hat{\mathbf{o}}$. We can thus form the likelihood for the observations $\mathcal{L}(\mathbf{o}|\mathbf{p})$, which is the basis for all model-fitting and analysis. For example a Bayesian fit would amount to minimizing

$$\mathcal{L}(\mathbf{o}|\mathbf{p})P(\mathbf{p}) \tag{3}$$

for some prior P on the parameters. Least-squares fitting is this minimization for the case of Gaussian likelihoods and uniform prior.

The first two classes here are $\mathbf{p}$, and $\mathbf{o}$ or $\hat{\mathbf{o}}$. They are functionally equivalent to simple arrays (`ArrayList` in the framework), with the exception that the `ParameterVector` can indicate that the observables are linear in some of the variables, which will speed minimization. We define these classes as trivial extensions of the array class just so that we have some kind of type-safing for the modelling routines.

```
class ParameterVector extends Noun {
public:
  void setElement(int i, double v); //access parameter i
  double getElement(int i);
  bool isLinear(int i);         //are observables linear in this parameter?
  int size();
}

class ObservableVector extends Noun;   // measurements to be fit
  void setElement(int i, double v); //access parameter i
  double getElement(int i);
  int size();
}
```

Next is an interface for a likelihood function $\mathcal{L}$. For fitting purposes we often need the derivative vector $\partial\mathcal{L}/\partial\hat{o}_i$. Fisher matrix calculations will need the second-derivative quantities $\partial^2\mathcal{L}/\partial\hat{o}_i\,\partial\hat{o}_j$.

```
interface Likelihood {
  double logLikelihood(ObservableVector o_meas, ObservableVector o_model);
  Vector logLDerivatives(ObservableVector o_meas, ObservableVector o_model);
  double secondDerivative(ObservableVector o_meas,
ObservableVector o_model,
int i, int j);
}
class GaussianLikelihood extends Noun implements Likelihood {
private:
  CovarianceMatrix covariance;
}
```

The Gaussian case implements the usual chi-squared

$$\mathcal{L} = \chi^2 = \mathbf{X}^T\Sigma^{-1}\mathbf{X}, \tag{4}$$

$$\mathbf{X} \equiv \mathbf{o} - \hat{\mathbf{o}}. \tag{5}$$

Here Σ is a `CovarianceMatrix`, which represents any positive definite matrix. More precisely, we may allow vanishing eigenvalues so we can represent degenerate parameter fits, and we can then also use this class to represent Fisher matrices that have no information in some dimensions. There is an interface

```
interface CovarianceMatrix {
  double getElement(int i, int j);
```

```
  void   setElement(int i, int j, double value);
  CovarianceMatrix inverse();
  double xCx(Vector x);
  ??? diagonalize
}
```

The implementations would likely include diagonal, block-diagonal, sparse, and general forms, each with its own `set()` methods and computationally optimized calculations.

The interface to a model that will be fit is next. In order to iterate the fit and determine parameter uncertainties or Fisher matrices, we will need the model derivative matrix $\partial\hat{o}_i/\partial p_j$.

```
interface Model {
  ObservableVector predict(ParameterVector p);
  Matrix parameterDerivatives(ParameterVector p);
}
```

Finally there are the classes that embody the fitting algorithms. These will conform to the style

```
class ModelFit implements Verb {
void fit(Model m,
         ObservableVector o,
         Likelihood l,
         ParameterVector p,
         CovarianceMatrix paramCovar);
}
```

This routine finds the `ParameterVector p` that maximizes the likelihood of the `ObservableVector o` under the given model and likelihood function. The input parameters, if any, are a starting point for the minimization. The output matrix is the covariance matrix for the fitted parameters. Implementations of `ModelFit` might include Marquardt-Levenberg, downhill simplex, linear, etc., with varying degrees of assumptions about the nature of the models and likelihood functions. We might have to put some flags in the interfaces about which models/likelihoods are linear, Gaussian, etc.

A Fisher analysis derives the optimal parameter covariances by taking likelihood derivatives about the input model (no measured data are required). The output is the *Fisher matrix*

$$F_{ij} \equiv \partial^2\mathcal{L}/\partial\hat{p}_i\,\partial\hat{p}_j. \tag{6}$$

The class to produce this is

```
class FisherAnalysis implements Verb {
CovarianceMatrix fisherMatrix(Model m,
                              ParameterVector p,
                              Likelihood l)
}
```

A further required method is a partial solution:

```
class PartialOptimization implements Verb {
void fit(Model m,
         ObservableVector o,
         Likelihood l,
         ParameterVector p1,
         ParameterVector p2,
 Vector          dLdp1,
         CovarianceMatrix paramCovar);
}
```

In this method, the parameters are divided into vectors $\mathbf{p}_1$ and $\mathbf{p}_2$. The fits finds the optimal $\mathbf{p}_2$ while holding $\mathbf{p}_1$ fixed at the input value, but on exit we need to know $\partial\mathcal{L}/\partial\mathbf{p}_1$ as well as the usual second derivatives. This method would be used to fit individual supernovae while retaining the solution's dependence on global quantities such as calibration constants.

4.5.1. How to Use It

To fit a model, one must create an `ObservableVector` and a `LikelihoodModel` that relects the uncertainties in the observables. Then one must create an implementation of `Model` that knows how to distribute the variables in the `ParameterVector` to all the physics models, run the necessary Verbs to predict the observables, and calculate derivatives w.r.t. parameters.

The initial `ParameterVector` guess is fed to the `ModelFit` Verb along with the information on model and observables. **???** How does the `Model` inform the `ModelFit` which parameters enter linearly into the observables? This is important for efficient fitting.

4.5.2. Special Cases

There are some special cases of Models which might be implemented. One common fitting case is where the observables are each the values of some function over the domain $\mathbb{R}^n$, and we make a series of measurements at specified points in $\mathbb{R}^n$. Examples would be the flux in a single band measured at a set of times t_i, or the brightness of some object sampled at a set of points (x_i, y_i) on the image plane. Then we might have a `Model`

```
class ModelFunction implements Model {
private:
   ArrayList samplePoints;
   Function  itsFunction;
public:
  void addSample(double x_i);
  int  nSamples() {return samplePoints.size();}
  ...
}
```

Here `itsFunction` is some function that maps from the sample domain $\mathbb{R}^n$ to the observable domain (which could also be multi-dimensional) and has a `ParameterVector` as well. The `samplePoints` array tells where in $\mathbb{R}^n$ we have measured data.

5. A General Scheme for Simulation of Astronomical Observations

The process of simulating observations of astronomical objects crosses several domains: Universe, Observatory, and Mission combine to create the simulated SourceData or PixelData. In many cases, the Analysis and Calibration routines are also simulating the observation of sources (in this case, to best model the SourceData), and should be able to make use of the same architecture.

An observation begins with an `AstronomicalSource`, which is a source of photons. The `AstronomicalSource` has a luminosity `SpectralFunction` and an intrinsic 2d `Shape`. The photons then pass through any number of media which impart `PropagationEffects` on the beam:

Extinction The source photons can be removed from the beam by the medium. A `transmission()` member function gives the `SpectralFunction` of the fractional transmission.

Emission A diffuse background of photons can be added in the vicinity of the beam. The `emission()` method gives this background.

Distortion The photons can be deflected, remapping the position and distorting the shape of the source.

Blur The images can be blurred by the medium. The `blur()` method returns a `PSF` that describes this.

Not all media have all of these effects, but we can incorporate many relevant effects in this framework: host dust and background light; intergalactic and Galactic dust; zodiacal light; the atmosphere; optical systems; and the detector itself. `PropagationEffects` that do not distort the image, for example, can return a null on the `distort()` method. Finally, an observation terminates at a `Detector`, which includes information on pixelization and a noise model in addition to the other attributes of a `PropagationEffect`. This ordered chain of objects: `AstronomicalSource`, any number of `PropagationEffects`, and a `Detector`, constitutes an `Observation` (with a few additional quantities as specified below).

The division between Universe properties and Observatory properties will normally be at the top of the atmosphere (or before entrance of the spacecraft aperture). As illustrated in Figure 1, the class `Target` defines the source and the intervening media that belong the the Universe. The final element of this chain will normally be the `ZodiacalDust`. The difficulty here is that when the Universe is constructing `Targets`, it does not know the ecliptic longitude and solar elongation at the time of observation, so it cannot fully specify this `PropagationEffect`. The solution is a new class, the `PropagationFactory`, which is capable of producing a `PropagationEffect` when, later in the simulation, the circumstances of the observation are determined. In this case, the `PropagationFactory` implementation is `ZodiacalLight`; it can produce on demand an instance of `ZodiacalSightLine`, which is an implementation of `PropagationEffect`. In general, a `PropagationFactory` can produce a `PropagationEffect` once informed of the `ObservationCircumstances`.

The part of the propagation chain that belongs to the Observatory is called a `Channel`. The included `PropagationEffects` will likely include the telescope itself, any reimaging optics or filters, and, for ground-based observations, a `PropagationFactory` implementation called `Atmosphere`. The `Channel` chain is terminated by the `Detector`.

The job of the Mission domain is to take the `Target` list from the Universe and the `Channel` list for a given exposure and produce an `Observation` object for each `Target` that is within the field of view of the `Channel`. Part of this task is to determine the `ObservationCircumstances` for the exposure, and use them to replace any `PropagationFactorys` that are in the chain with the `PropagatinEffects` that they produce.

There are three additional constants that are necessary to complete the specification of the `Observation`. First is the angular diameter distance $D_A(z)$ to the source.[2] This is obtained from the `Cosmology` object given the redshift of the `AstronomicalSource`. D_A is needed to convert the `luminosity()` of the source into a flux, and to convert the domain of the source `Shape` from physical units (kiloparsec) to angular units (rad or arcsec).

The other two constants reside in the `Channel` structure: one is the collecting area of the telescope, which converts the incident flux into a photon rate. The second is the focal length onto the detector, which converts all the angular units of `Distortion` and `Shape` to/from the physical dimensions of the `Detector`.

The structure of an assembled `Observation` is described in Figure 2, and a version with sample

[2]Or simply the distance d for non-cosmological sources.

implementations attached to the interfaces is shown in Figure 3.

Note the "Stack Marker" in the `Observation`, which is just a null `PropagationEffect` placed in the sequence to mark an important location, such as the top of the atmosphere. This will be helpful because the Calibration will always need to be defined relative to some reference plane.

6. Universe Domain

6.1. Cosmology

Describes the zeroth-order geometry of the Universe.

```
interface Cosmology {
  // generic methods for changing cosmological parameters
  void   setParameters(ParameterVector p);
  ParameterVector getParameters();

  double dA(double z);    //angular diameter distance
  double dL(double z);    //luminosity distance
  double dVdz(double z);  //volume element
  double lookback(double z); //lookback time
  double H(double z);     //Hubble parameter vs z

  Vector dDAdp(double z);  //derivatives w.r.t. parameters
  Vector dDLdp(double z);  //derivatives w.r.t. parameters

  // others for linear growth factor, etc.???
}
```

One implementation is of course our usual cosmology with parameters $\{\Omega_m, \Omega_X, w, w_a, h\}$.

???Should the `Cosmology` also be responsible for inhomogeneities (lensing), so that the above calls take a `SphericalCoordinate` as argument as well?

6.2. Astronomical Sources

Any source of photons in the Universe. It is presumed that each implementation is a Noun, and the `iName` of sources serve as logical IDs and it is hence desirable to make them unique within a given Run.

```
interface AstronomicalSource {
```

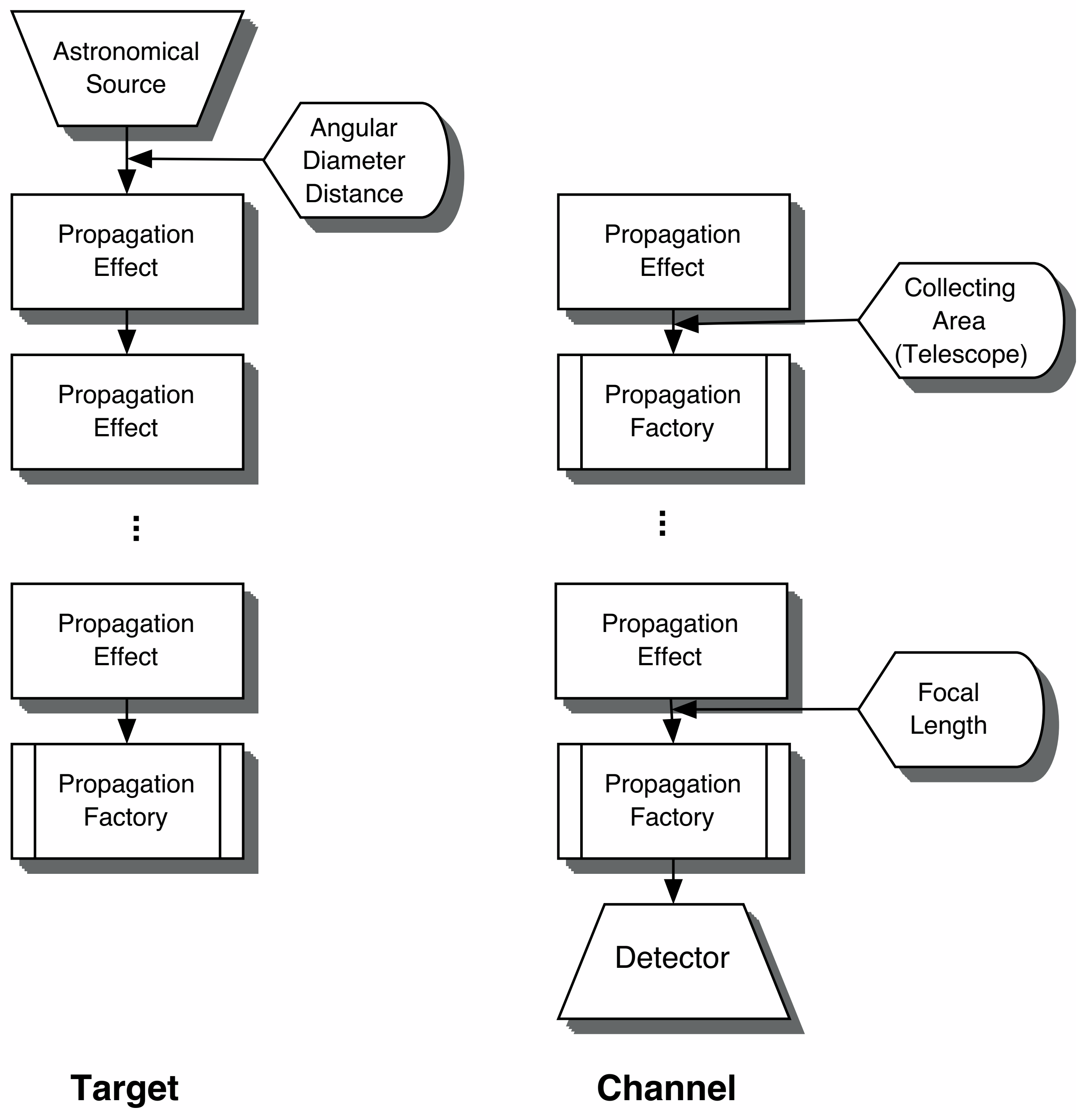

Fig. 1.— The `Target` and `Channel` structures assembled by the Universe and Observatory, respectively, are shown. Photons flow from the top down, originating in an `AstronomicalSource` and passing through various `PropagationEffect`s to a `Detector`. Some of the intervening effects are not fully determined until the details of the observation are known; their places are held by `PropagationFactory` objects.

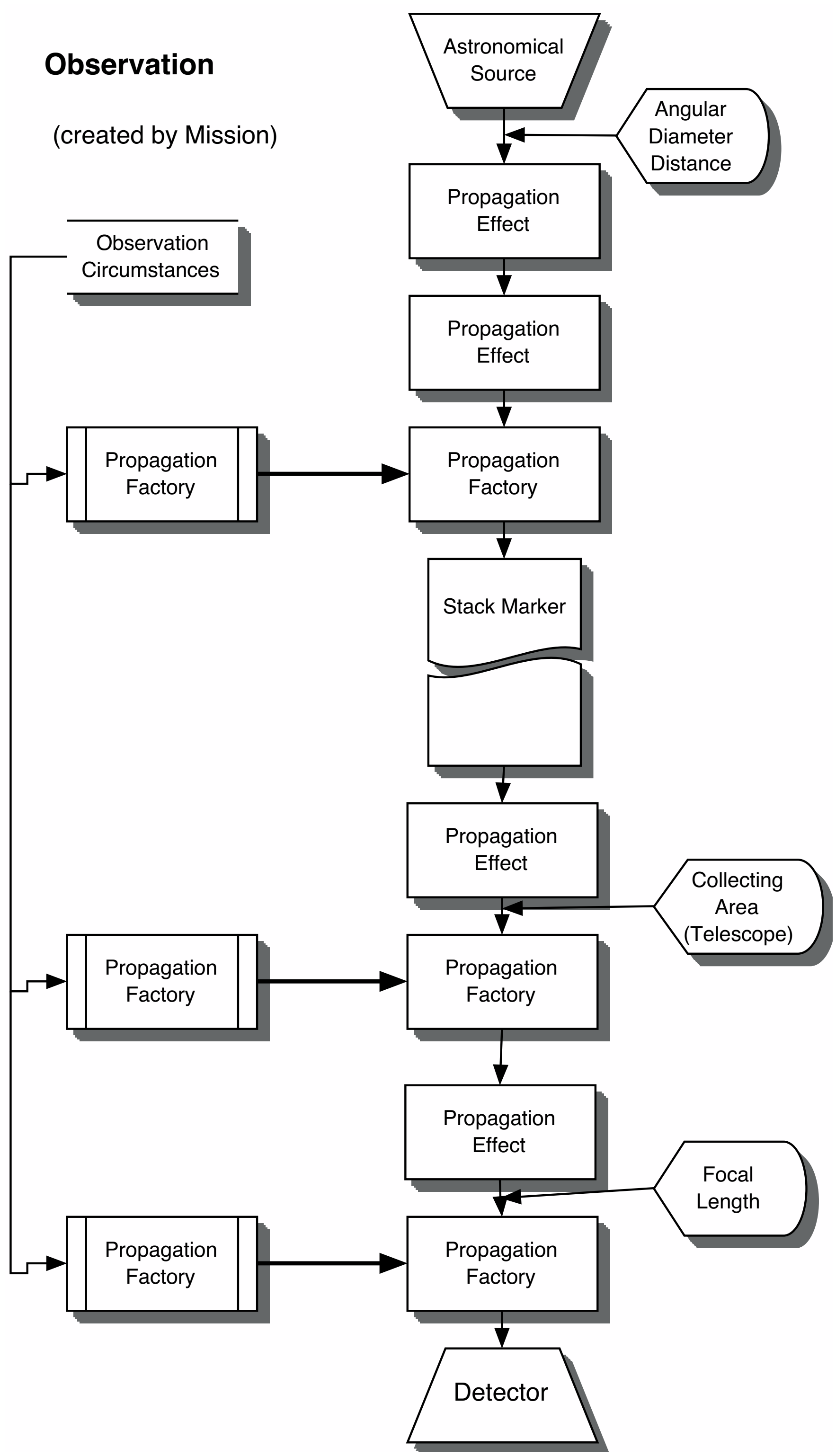

Fig. 2.— The `Observation` object is assembled in the Mission domain by attaching a `Target` to a `Channel`. The `ObservationCircumstances` are given to each `PropagationFactory` so that the correct `PropagationEffect` may be put in the chain.

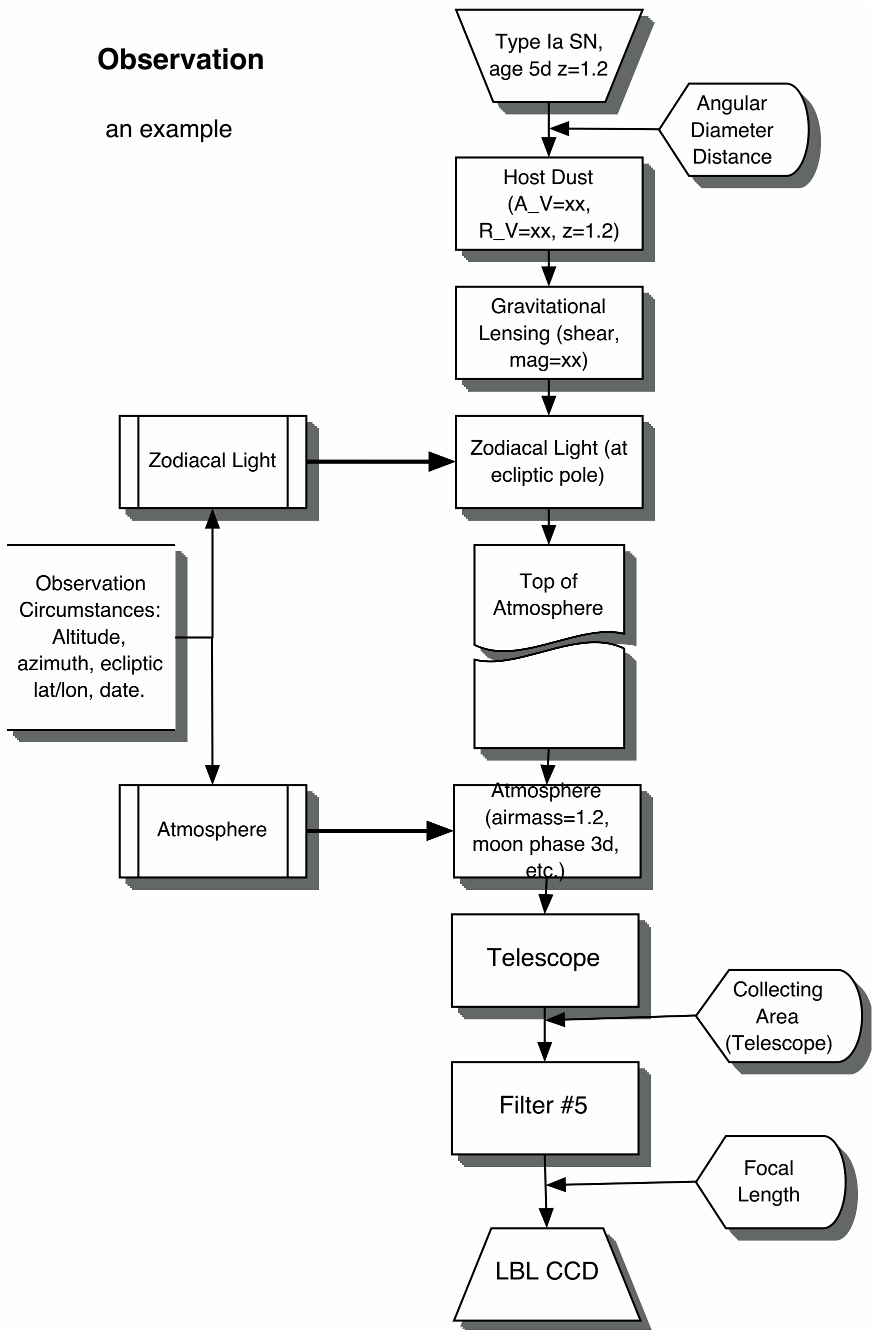

Fig. 3.— Another view of a `Observation`, this time with names in each block showing the actual physical effect being modelled in the implementations.

```
  // generic methods for changing model parameters
  void   setParameters(ParameterVector p);
  ParameterVector getParameters();

  SpectralFunction luminosity(UT t); // rest-frame luminosity
  SpectralFunction dLdp(UT t, int iParam);  // luminosity deriv w.r.t. parameters

  SphericalCoordinates  position(UT t);  //position before any distortion
  double redshift();   // cosmological redshift (if cosmological, 0 otherwise)
  double losV(UT t);   // rest-frame line-of-sight velocity
  CartesianCoordinates position(UT t);  // distance and velocity for nearby
  CartesianCoordinates velocity(UT t);  // sources, like stars & planets.

  // Some helpful hints:
  bool isVariable();     //Does luminosity vary w/time
  bool isMoving();       //Does (unlensed) position change with time

  Shape  itsShape(double lambda);   //Shape (in proper units)
}
```

??? A note on Dates: I suggest a convention that the date input to the `AstronomicalSource` (or indeed to any Universe class) be the date at which the light arrives at the Solar System barycenter. The `AstronomicalSource` is then responsible for calculating the proper time at the source, which it can do because it knows the redshift z. A complication is that gravitational lensing introduces time delays; but only in very rare cases will we care about this, such as when simulating multiply-imaged sources, and we can plan perhaps to worry about these cases individually.

6.2.1. Shapes

Note the introduction of another class that describes a map of surface brightness over 2d angular or physical variables:

```
interface Shape {
  complex kValue(double kx, double ky);
  complex xValue(double x, double y);

  // Some hints for usage of this Shape:
  double  maxK();          //largest significant k-vector
  double  stepK();                        //suggested k-space resolution
  bool    isAxisymmetric();
```

```
  // additional methods for "draw" onto an image, returning
  // particular moments, etc.
}
```

Expected implementations of `Shape` include

```
class PointSource extends Noun implements Shape;
class GaussianEllipse extends Noun implements Shape;
class ExponentialEllipse extends Noun implements Shape;
class DeVaucouleursEllipse extends Noun implements Shape;
class Shapelet extends Noun implements Shape;
class TabulatedShape extends Noun implements Shape;
```

There will probably have to be a Verb that can convolve a `Shape` with a `PSF` to return another `Shape`:

```
class ConvolveShape implements Verb {
  Shape convolve(Shape source, PSF thePSF);
}
class DistortShape implements Verb {
  Shape distort(Shape source, Distortion dist);
}
```

6.2.2. Supernova Classes

A base class for all supernova would be:

```
class Supernova extends Noun implements AstronomicalSource {
public:
  UT dateOfExplosion();
  bool isVariable() {return true;}
  bool isMoving() {return false;}
  Galaxy itsHost() {return hostGalaxy.target()};
  Shape itsShape(double lambda) {return PointSource;}
private
  Link hostGalaxy;
}
```

??? Note use of the SNAPSim `Link` class here to hold a pointer to the host `Galaxy` object. The syntax used here for extracting the target of the link is a guess at what SNAPSim will provide.

For most of this document I have glossed over the distinction between `Link` and `Composition` relationships.

Type Ia supernovae do not in general have any methods or data beyond those of this `Supernova` class. But we may choose to implement, as in SNAPfast, models of TypeIa that predict some intermediate observable quantities other than just the `luminosity`, *e.g.* light-curve parameters such as stretch, rise time, etc. So there might be an additional class (or an abstract class with several implementations) called

```
class SupernovaIa extends Supernova {
  ... // Implements all the methods of AstronomicalSource() and
  ... // Supernova()

  ObservableVector observables();  //intermediate observables
  ObservableVector dObsdp(int i);  //and their derivs w.r.t. parameter i
}
```

6.2.3. Galaxies

Galaxies are `AstronomicalSources` with finite extent. The spectrum of a galaxy would likely be specified by some set of star-formation and extinction parameters. Note that `luminosity` should return the total luminosity of the entire galaxy, `itsShape` returns the distribution of that luminosity across the sky at a chosen wavelength.

??? Note that a `Galaxy` will need to know the `Cosmology` somehow in order to calculate its `luminosity`, because the stellar-evolution codes will need to know the age of the Universe at the relevant redshift.

```
class Galaxy extends Noun implements AstronomicalSource {
public:
  bool isVariable() {return false;}
  bool isMoving() {return false;}
  ??? some kind of connection with a catalog entry
}
```

Ultimately there will be a

```
class AGN extends Noun implements AstronomicalSource {
public:
  bool isVariable() {return true;}
  bool isMoving() {return false;}
```

```
  Galaxy host() {return itsHost.target();}
private
  Link itsHost;
}
```

6.2.4. Stars

Assuming here that all stars are too close to have comsological redshifts, there is a base class that looks like

```
class Star extends Noun implements AstronomicalSource {
public:
  Shape itsShape(double lambda) {return PointSource;}
  double redshift() {return 0.;}
private:
  double distance;
  ??? proper motion specification
}
```

Some stars are variable, some aren't. We'll assume that `isMoving()` is true only if there is non-zero proper motion; parallax does not count as "moving."

6.2.5. The Phantom Source

A special-purpose source is one which emits no photons. It allows us to use the framework to easily construct maps of the background light.

```
class PhantomSource extends Noun implements AstronomicalSource {
public:
  Shape itsShape(double lambda) {return PointSource;}
  double redshift() {return 1000.;}
  bool isVariable() {return false;}
  bool isMoving()   {return false;}
  SpectralFunction luminosity(UT t) {return ScalarSpectralFunction(0.);}
  ... etc.
}
```

6.3. Propagation Effects

Any medium that lies between the AstronomicalSource and the Detector may be derived from

```
class PropagationEffect extends Noun {
public:
  void   setParameters(ParameterVector p);
  ParameterVector getParameters();

  SpectralFunction transmission();        //Transmission of bgrnd photons
  SpectralFunction emission();          //surface brightness of added photons
  Distortion distortion();   //distortion of bgrnd image
  PSF blur();                           //blurring of bgrnd image

  // This one needed for modelling:
  SpectralFunction dTdp(int i);           //Deriv of transmission  w.r.t. parameter
}
```

In general an intervening PropagationEffect can have four effects on our view of its background: absorbing photons; adding photons; distorting the image; or blurring the image. Galactice dust, for example, has important extinction but no significant visible emission, distortion, or blurring, so it might be realized as a PropagationEffect that has null returns for emission(), distortion(), and blur(). The atmosphere is probably significant in all four aspects (this will be in the Observatory domain, not Universe).

Each individual observation of an AstronomicalSource will be done through a series of PropagationEffects, the effects of which can be compounded because of our ability to generate products of SpectralFunctions, Distortions, and PSFs.

The important PropagationEffects in the Universe domain will be:

```
class HostGalaxy extends Noun implements PropagationEffect;
class IntergalacticDust extends Noun implements PropagationEffect;

class GalacticDust extends Noun implements PropagationEffect;
class GravitationalLensing extends Noun implements PropagationEffect;

class ZodaicalLight extends Noun implements PropagationFactory;
class ZodaicalLineOfSight extends Noun implements PropagationEffect;

class TopOfAtmosphere extends Noun implements PropagationEffect;
```

Note that the Zodiacal light, has a `PropagationFactory` because the exact realization of `PropagationEffect` is indeterminate until the circumstances of the observation are known (§5). All of these effects may be considered to have null `blur()` in the visible/NIR. Non-null characteristics include:

`HostGalaxy` has `transmission()` as a `Dust` screen at the redshift of the host. The `emission` is the surface brightness of the host galaxy.

`IntergalacticDust` has `transmission()` specified by some model of grey dust and the redshift of the source.

`GalacticDust` has `transmission()` as a `Dust` screen at zero redshift.

`GravitationalLensing` has only `distortion()`, which is determined by the source redshift and position.

`ZodaicalLight` has only `emission()`, which is determined by the solar coordinates of the observation.

`TopOfAtmosphere` This is just a placeholder entry. All of the methods return null, but it allows us to mark the point in the stack of `PropagationEffects` which divides the Universe from the Observatory. It is the job of the Calibration domain to simulate the spectral response of elements past this marker.

Classes that will certainly be needed to implement the astronomical `PropagationEffects` are

```
// Clayton-Cardelli-Mathis model of extinction:
class CCMDust extends Noun implements SpectralFunction {
private:
  double Av, Rv;    // A_V and R_V of dust
  double z;         // Redshift of the dust screen
}

// Greg Aldering's model for Zodiacal emission spectrum
class ZodiacalLight extends Noun implements SpectralFunction {
private:
  double solarElongation;
  double eclipticLatitude;
}

// A model for grey dust absorption
class GreyDust extends Noun implements SpectralFunction {
private:
  double sourceZ; // redshift of source
```

```
  double ???             // parameters of the dust distribution model
}
```

Here is the interface for a `PropagationFactory`:

```
interface PropagationFactory {
  PropagationEffect realize(ObservationCircumstances oc);
  bool ifAbsorbs();
  bool ifEmits();
  bool ifDistorts();
  bool ifBlurs();
  void appendTo(Target t);
  void appendTo(Channel t);
  void appendTo(NounArray t);
}
```

The boolean tests may be used to check if the observing circumstances will affect, for example, the apparent position or incident flux of this source. The `appendTo` methods tell this factory to append itself, or a realization of its `PropagationEffect`, to the list of intervening media maintained in a given `Target`, `Channel`, or each element of a `NounArray` of these.

6.4. Targets

For each `AstronomicalSource` we need to know the `PropagationEffects` through which it will be viewed. The `Target` structure contains both the source and the intervening media:

```
class Target extends Noun {
public:
  AstronomicalSource source;
  NounArray          propList; //??? can we use a list?
  double             dA;        //Angular diameter distance
  void               append(PropagationEffect pe);
  void               append(PropagationFactory pf);
  void               replace(??pointer to list member??, PropagationEffect pe);
  bool               isValid();

  SphericalCoordinates astrometricPosition(ObservationCircumstances oc);
  SpectralFunction   incidentFlux(ObservingCircumstances oc);
  // ??? a method to get observed Shape?
}
```

The isValid() method checks that propList contains only PropagationEffect and PropagationFactory entries. The astrometricPosition method calculates the apparent position of the object relative to the stellar reference frame. There is a dependence upon the date of observation for moving objects, and upon the position of the observer if the source is close enough to have significant parallax. This method will throw an exception if there is a propList element that has ifDistort()==true but cannot determine the distortion from the information present in the ObservationCircumstances structure. The incidentFlux() method similarly gives the incident flux (at top of atmosphere) from this source, which is just the product of the source luminosity and all the transmission() functions, divided by the distance factor.

6.5. Distribution Functions

Distribution functions tell us how to populate the Universe with AstronomicalSources and what PropagationEffects their photons will encounter before entering the Observatory. We first have an interface for generating a list of sources that might be seen over a given time period in a given part of the sky. **???** I think the distribution functions want to be Verbs, so that the run() method generates the desired target list. I'm not sure in Java if you can have a heirarchy of interfaces or if I need to make this an abstract base class.

```
class TargetGenerator extends Noun implements Verb {
public:
  NounArray generate(Cosmology c,
                     SolidAngle a,
                     UT startTime, UT endTime,
     UniformDeviate u);
}
```

The UniformDeviate object is a random number generator that is passed in. We assume a model in which the entire simulation run shares a single instance of the random number generator.

There can be various implementations of TargetGenerator, which make SNe, galaxies, stars, or perhaps jointly generate the galaxies and SNe so that the host relationships are physically based. Some implementations would have additional parameters, such as the min/max redshift of interest, or the faintest magnitude of galaxy to generate. A given realization of the Universe can have one or more implementations of TargetGenerator instantiated—more than one is appropriate if there are multiple populations of objects on the sky. There should be a master list of all active TargetGenerators:

```
class MasterGenerator extends Noun implements Verb {
public:
```

```
  NounArray sourceGenList;
  NounArray propagationFactoryList;
  NounArray generate(Cosmology c,
                     SolidAngle a,
                     UT startTime, UT endTime,
     UniformDeviate u);
}
```

The `generate()` method of this class just calls `generate()` for all of the `TargetGenerator`s in the `sourceGenList`.

It is also necessary to realize the `PropagationEffects`, for example create a lensing or extinction map on the sky. The previously defined class `PropagationFactory` is appropriately reused here. It is the job of a `PropagationFactory` to add a new `PropagationEffect` to the `propList` of the `Target`s it affects. The `PropagationFactory` may instead add itself to the `propList` if there is a need for later information about `ObservationCircumstances`. For example there is

```
class SchlegelDustMap extends Noun implements PropagationFactory {
public:
  PropagationEffect generate(SphericalCoordinate s);
}
```

which can produce a `CCMDust` extinction law for the chosen coordinates based on the Schlegel/Finkbeiner/Davis map, and package it into a `PropagationEffect`. The `appendTo()` methods of this particular factory would ask the `Target` for its `astrometricCoordinates` in order to determine the line-of-sight extinction. This determination is independent of `ObservationCircumstances` (as long as our observatory stays inside the solar system!).[3]

6.6. The Universe Procedure

The end product of the Universe is a `NounArray` of `Targets` (should we make a `TargetList` class???). To produce this, the steps are:

1. Create the parent `Cosmology` for the simulation.
2. Create a `SourceGenerator` for each set of objects to inhabit the Universe.
3. Create a `MasterGenerator` and add each `SourceGenerator` to its `sourceGenList`.

[3] A detail: the Galactic extinction depends upon distance for objects within the Galaxy.

4. Create a PropagationFactory for each intervening medium (excluding those, such as host galaxies, that will be generated by the SourceGenerator). Add each to the propagationFactoryList of the MasterGenerator. Note that the order will matter here.

5. Determine the bounds of the survey (from Mission domain).

6. Call the generate() method of the MasterGenerator with the desired survey bounds (this is also its run() behavior). It will first generate Target lists from each SourceGenerator and combine them all into one list. Then it asks each PropagationFactory to appendTo() every Target.

The Target list is now complete. Optionally the list can be assembled by client calls directly to the generator classes, bypassing the MasterGenerator.

7. Observatory Domain

7.1. Overall Scheme

The Observatory domain specifies everything from the top of the atmosphere or observatory input to the detector outputs. All of the instrumentation and environmental effects ahead of the detector itself will be described as PropagationEffects, culminating in a Detector class that describes the conversion from photons into electrons on a two-dimensional grid. One of the PropagationEffects is the Telescope that has an extra bit of information: the collecting area. A Channel is list of PropagationEffects, one of which is the Telescope, and a Detector, completely describing the path of one set of photons. The only additional bit of information needed for a Channel is the focal length, which gives the plate scale for conversion of physical detector pixel size into an angular scale.

A given observatory comes equipped with a multiplicity of possible Channels. A Configuration of the observatory is a list of Channels that operate simultaneously. Each Detector knows its position in the focal plane (in meters) relative to the optic axis. A Configuration for SNAP would of course contain one Channel for each of the imaging arrays, which do not overlap in the focal plane. A convenient way to represent the image-slicing spectrograph is as an array of Channels that all occupy the same physical location on the focal plane, but each have slightly different SpectralFunctions.

The Mission domain will contain a class Exposure which will specify a single exposure taken with the observatory in a chosen Configuration. The Exposure also will specify the start/stop time of the exposure and the Orientation of the observatory during the exposure. An Exposure should therefore contain or link to everything that there is to know about the functioning of the observatory for a given period of time.

??? Need a way to specify the jitter contribution to the PSF. Perhaps as a propagation effect? Or somewhere in the Exposure?

7.2. Telescope and Optics

The `Telescope` is derived from a `PropagationEffect` because it can have a PSF, transmission function, distortion, and perhaps even some thermal emission. But it also has a collecting area:

```
class Telescope extends PropagationEffect {
public:
  double collectingArea();
  ??? need a diameter() method?
}
```

The zodiacal light, atmosphere, filters and other optical elements can be implementations of `PropagationEffect`, each created by an appropriate `PropagationFactory`. In particular we will expect to have the two classes:

```
class Atmosphere extends Noun implements PropagationFactory {
public:
  AtmosphereLineOfSight realize(double zenithAngle, UT date)
  ??? also will depend on observatory altitude and position?
}
class AtmosphereLineOfSight extends PropagationEffect {
public:
  double airmass;
  UT     date;
  double moonPhase;
  double moonAngle;
  double r0; //Atmospheric turbulence scale length
  ????
}
```

which will tell us the properties of the atmosphere in a given direction at a given time.

7.3. Detector

The `Detector` is also derived from `PropagationEffect` as it has a `SpectralFunction` QE and a `PSF` that describes the pixel response function [??? This PSF needs to be converted from distance units to angular units using a focal length.]. The `Detector` has a few additional attributes:

```
class Detector extends PropagationEffect {
public:
  double pixelSize;        //pixel size, in meters
  int    xSize;            //size of array, x dimension
  int    ySize;            //size of array, y dimension
  double xPosition;        //Position of detector center in focal plane (m)
  double yPosition;
  NoiseModel noise();
```

It is assumed that the detector is physically rectangular, and is centered at the given position relative to the optical axis of the telescope.

A `NoiseModel` describes the variance of the output of a given pixel's detection process. The interface is

```
interface NoiseModel {
  double variance(double countRate,
                  double exposureTime);
  double realize(double countRate,
                 double exposureTime);
}
```

A common implementation is

```
class CCDNoise extends Noun implements NoiseModel {
public:
   double readNoise;
   double darkRate;
   double variance(double countRate,
                   double exposureTime) {
      return readNoise*readNoise + exposureTime*(countRate+darkRate);
   }
   double realize...;
}
```

??? Missing from the domain is any way to describe the dead times for readout and moving the telescope.

7.4. Aggregate Classes

A complete chain of the Observatory is a `Channel`:

```
class Channel extends Noun {
public:
  NounArray propList;
  Detector  itsDetector;
  double    focalLength;
  double    collectingArea();  //find this from the Telescope entry

  bool      isValid();

  SpectralFunction efficiency(ObservingCircumstances oc);
  SolidAngle fieldOfView(ObservingCircumstances oc);
  bool      isVisible(SphericalCoordinates c, ObservingCircumstances oc);
}
```

The `isValid()` method returns true if all the elements of the `propList` are either `PropagationEffects` or `PropagationFactorys`, and if exactly one of them is a `Telescope`. The `efficiency()` function gives the end-to-end QE vs wavelength, *i.e.* the product of all the `transmission()` functions of its elements.

The `fieldOfView()` function returns a rough outline of the region of sky subtended by the detector under the given observation circumstances. The returned region should contain the entire observed area, but is allowed to be a slightly larger region if this allows the test to be faster (for example by using the circumscribed rectangular area). The `isVisible()` function is a more exact (and possibly slower) test of whether a given sky coordinate falls onto the detector. For both of these methods we are essentially propagating the (rectangular) outline of the `Detector` backwards through all the distortions of the optics to find its outline on the sky.

An available set of simultaneously active `Channels` is a

```
class Configuration extends Noun {
public:
  NounArray channelList;
  bool      isValid();
}
```

The Boolean just checks that each element on the list is a link to a `Channel`.

??? All the `Channels` of a `Configuration` are assumed below to share a common set of exposure times. This would not describe well the situation where we are simultaneously running the spectrograph and the imager, but with different exposure times. Nothing in the framework here precludes, however, the possibility of having more than one `Configuration` in use at a given time (indeed this might happen with multiple ground-based telescopes for instance). For now we leave it up to the individual programmer to insure that an observatory is not specified to be used in an impossible fashion, *e.g.* two different pointings for SNAP at the same moment.

8. Mission Domain

Data structures in this domain specify the schedule of observations to be made, the dither/repetition patterns of the exposures, and the `Observation` structure that combines the source information from Universe domain with the Observatory information for all the detectors that see the source.

The principle Verbs of this domain are an algorithm for determining an exposure schedule—which is likely to be customized for every possible scheduling algorithm—plus a process for creating the `Observation` list, which is likely to be used in the same way for nearly all simulations.

8.1. Data Structures

The following Nouns (and manipulation functions) are needed:

```
class ExposureSequence extends Noun {
public:
  UT     startTime;
  double exposureTime;    //Per exposure
  int    xDitherCount;
  int    yDitherCount;
  double xDitherStep;     //in radians
  double yDitherStep;
  ??? allow a list of Orientations as alternative to grid dither?
  int    nRepeat;         //number of exposures at each dither posn
  Link   itsConfiguration;  //link to an observatory Configuration
  Orientation pointing;
  ??? something about dead times - array of exposure start times??
}

class ObservationCircumstances extends Noun {
public:
  UT     startTime;       //barycentric time
  UT     endTime;
  SphericalCoordinatesICRS  lineOfSight;
  CartesianCoordinatesICRS  observatoryPosition;
  CartesianCoordinatesICRS  observatoryVelocity;  //for stellar aberration
```

```
  CartesianCoordinatesICRS  sunPosition;
  CartesianCoordinatesICRS  moonPosition;
  CartesianCoordinatesICRS  earthPosition;  //the geocenter
  double                    horizonElevation;  //where is horizon/limb of Earth
  ??? is this all that's needed to specify any PropagationEffect?
}

// Some functions that return commonly desired quantities:
// (???could be methods of the Noun)
double sunAltitude(ObservationCircumstances oc);
double moonPhase(ObservationCircumstances oc);
double zenithAngle(ObservationCircumstances oc);
double airmass(ObservationCircumstances oc);
double azimuth(ObservationCircumstances oc);
double solarElongation(ObservationCircumstances oc);
double eclipticLatitude(ObservationCircumstances oc);
bool isVisible(ObservationCircumstances oc);  // does Earth obstruct source?

class Observation extends Noun {
public:
  AstronomicalSource source;
  NounArray          propList; //??? can we use a list?
  Detector           detector;
  double             dA;        //Angular diameter distance
  double             collectingArea();  //get this from the Telescope
  double             focalLength;
  void               replace(??pointer to list member??, PropagationEffect pe);
  bool               isValid();
  SphericalCoordinates astrometricPosition();
  double             xPixelPosition, yPixelPosition;

  double sourceRate(); //source count rate (per s)
  double skyRate();     //background count rate (per pixel per s)
  PSF    thePSF();      //cumulative ePSF for source (in angular units)

  Link   itsTarget;
  Link   itsChannel;
  Link   itsExposureSequence;

  double dRate_dParam(??index of effect??, int i);
```

```
}
```

The last method here is for the purposes of adjusting model `Observation`s to the data. It gives the derivative of the output count rate with respect to parameter i of a chosen element in the source/propagation/detector chain.

??? Where does cosmic-ray information come from?

8.2. Verbs

The first function of Mission domain is to determine the schedule for the observations. This is accomplished by a

```
interface MissionAlgorithm {
  NounArray schedule(NounArray targetList,
     NounArray configurationList);
}
```

This method (which is also the `run()` for this Verb) takes as input the list of `Target`s in the Universe, along with the available `Configuration`s of the Observatory. The output is the set of `ExposureSequence`s to be taken during the mission.

The implementations of this interface can become quite complex if they are modelling any kind of adaptive targeting algorithm, because it might involve calling other Mission-, SourceData-, and Analysis-domain processes to mimic the collection and processing of data that would be done in deciding how to trigger spectroscopy. Note that an implementation that is doing such an algorithm test would likely ignore the input `targetList` because it contains *truth* inputs that are unavailable for the real data.

A simpler implementation, such as needed for SNAPfast, simply has an open-loop scheduling algorithm, that can assume we know which Ia's are interesting and points the telescope at them without worrying about the trigger mechanism.

A Verb (**???** or just function?) that will be usefully used by implementations of `MissionAlgorithm` is one which generates the `Orientation` of the Observatory that is necessary to put a chosen `Target` at a chosen location on a chosen `Channel`'s detector:

```
class PointAtTarget implements Verb {
  Orientation pointAt(Target t,
                      Channel c,
                      double xPixel, double yPixel,
                      double positionAngle,
```

```
                      UT     time);
}
```

There should be either a Boolean flag or a thrown exception to indicate when the target is not observable at the chosen time.

The second major method of Mission is to take the list of `Target`s and the list of `ExposureSequence`s, and determine which of the former are observed by each `Channel` of the Observatory during each of the latter.

```
class Observe implements Verb {
public:
  NounArray run(NounArray targetList,
                NounArray exposureList);
}
```

The return is an array of `Observations`, one for each time each `Target` is seen by a `Detector` during an `ExposureSequence`. This completes the task of the Mission domain.

9. PixelData Domain

PixelData domain contains everything that deals with pixelized image data. The simulation effort will not be concerned for some time with the details of bias subtraction and flat-fielding, so we will simply be operating with calibrated images. The simulation architecture will provide us with an Image class, so there are few or no additional data structures required in this domain.

There are two classes of Verbs that must be produced for this domain. The first is

```
class Render implements Verb {
public:
  Image run(NounArray        observationList,
    UniformDeviate   random,
            ExposureSequence whichExposure,
    int              whichXDither,
    int              whichYDither,
    int              whichRepeat,
    CosmicRayModel   cr);
}
```

There is some calling syntax (??? not really decided here) that tells this Verb which individual exposure of which `ExposureSequence` is to be simulated. We assume that each `Channel` generates an individual image as each contains a single `Detector`. The `random` input is needed to realize the noise on the image. The steps for this Verb are

1. Create an appropriately sized empty `Image.`
2. Find which `Observations` in the input list refer to the desired exposure.
3. Use the `Shape`, `Distortion`, and `PSF` information contained in the `Observation` to determine how to draw the (noiseless) object onto the detector pixel grid.
4. Use the `countRate()` and `exposureTime` information in the `Observation` to normalize the total received flux from the source, and sum this source into the pixelized image.
5. Use the `NoiseModel` of the `Detector` to realize measured pixel values for this image. **???**The sky rate will have to be known for all pixels but right now it is carried in individual `Observations`—need some way to extract a image-wide sky level, perhaps using one or more `PhantomSources`.
6. Use the `CosmicRayModel` to add cosmic-ray hits to the image.

??? There might be some `Combine` Verb to produce a combined image from a dithered set of exposures of the same field. Alternatively the `Render` Verb might be instructed to construct a single combined image for an entire `ExposureSequence` rather than bothering with the individual frames.

The second major class of Verbs in this domain will be those that examine an `Image` or series of `Images` to produce `SourceData` lists of all the objects detected. For instance one might be a `SExtractor` kind of process. There will likely be many such things developed once we start simulating image algorithms. **???** I postpone for now any attempt to enumerate them all, since we will initially skipping the PixelData entirely.

10. SourceData Domain

SourceData contains the integrated quantities of observed targets that will be extracted from single (or dithered sets of) images. The canonical example is the instrumental flux of a point source that is extracted from an image by some photometry algorithm.

For a point source there are only three observables (flux and two position components) on a single image. The following class saves these quantities, as well as the covariance matrix for them. The `fluxVariance` method provides the flux variance after marginalization over the position.

```
class PointSourceData inherits Noun {
  Link              itsChannel;
  Link              itsExposure;
```

```
  double             flux();
  double             fluxVariance()
  SphericalCoordinates position();
  ObservableVector   observables();
  CovarianceMatrix   covariance();
}
```

The flux is the count rate on the detector. It requires some Calibration-domain classes to be converted an estimate of the flux in physical units. **???**Perhaps each `Channel` can include a nominal conversion factor from count rate to flux, so that we can express the `flux()` here is appropriate units.

Likewise the measured position may require some massaging by Calibration classes in order to yield the true best estimate.

An assumption made here is that there is no covariance between quantities in different SourceData instances. Covariances between source fluxes are usually generated only via calibration parameters, so these uncalibrated data should be independent. An exception is for objects that share pixels (and hence variance) on an image, for instance a SN its host, or crowded objects. In the future it may be necessary to have catalogs that include more generally coupled observables using the classes described below. For now we will assume that our SourceData catalogs consist of decoupled observables.

For resolved galaxies, there is additional `Shape` information in the SourceData. The `observables` and `covariance` methods now include some set of shape parameters. There are `size` and `ellipticity` calls, that will be characteristics of any implementation of `Shape`. **???** Where is the measured shape combined with PSF estimation to give an intrinsic-shape estimate? If determination of the PSF map is part of going from PixelData to SourceData, then there might be covariance among SourceData due to uncertainties in this.

```
class ResolvedSourceData inherits Noun {
  Link               itsChannel;
  Link               itsExposure;
  double             flux();
  double             fluxVariance()
  SphericalCoordinates position();
  Shape              intrinsicShape();
  ObservableVector   observables();
  CovarianceMatrix   covariance();

  double             size();
  void               ellipticity(double& e1, double& e2);
```

```
}
```

There must be some Verb that calls the ETC to generate a FluxPoint for each Observation of each point-source object. For extended objects, the ETC must also measure shape parameters, etc.

??? How will we handle combining info from different exposures? Could propagate the covariance matrix for each individual exposure, and later collect all exposures and fit for single best-fit position/flux/shape.

10.1. Truth and realization

10.2. Catalogs

10.3. ETC

11. Calibration Domain

The task of the calibration is to create a model for the response of the instrument. Typically the "instrument" is taken to be everything from the top of the atmosphere down (or from the entrance aperture of a space telescope). In essence the Calibration domain thus contains, for each `Channel` of the detector, a little model of how the positional and flux outputs of that `Channel` are related to the position and flux of the celestial object—in other words, a `Distortion` plus a `SpectralFunction` that describes the absolutely-calibrated effective collecting area of the `Channel`.

11.1. Model Channels

11.2. Standard Bands

11.2.1. Correlated data sets

11.3. For discussion with Calibration group

[**NOTE:** The following expository material is for discussion purposes, probably won't go into the actual specification document.]

The simulation code in the Observatory domain contains information that fully describes the instrument response. But this *truth* information should not be available to the Analysis domain because when analyzing the real data, we will not have perfect *a priori* knowledge of the instrument response. We will have to create a model, using the data itself, information from ground tests, and calibration observations with the telescope. So the Calibration model of the instrument, unlike the

Observatory specifications, have free parameters that have to be adjusted to give the best estimate (and uncertainties) that agrees with all calibration information.

Flux calibrations can be used two different ways. One is to have a model for the instrument's spectral response. A posited input flux spectrum (*i.e.* at the top of the atmosphere) can be convolved with the response model to *predict* the flux in the `SourceData`, which are our instrumental outputs. Most rigorously we then view the analysis as a process of adjusting the free parameters in the source models *and* in the calibration models until we obtain best agreement with the collection of `SourceData` from all the science and calibration observations. Let's call this Method A.

Typical astronomical calibrations often try to do something different, let's call it Method B: they posit a set of *standard bandpasses* and try to come up with a conversion from the instrumental outputs into fluxes through the posited standard bandpasses. The typical photometric calibration equation takes intrumental magnitudes in one or more bands (plus perhaps engineering data such as airmass), maybe some prior info on the spectrum/color of the target, and outputs an estimate of fluxes in standard bands/colors.

Method A is preferred for analysis of the experiment's data. Ultimately we want a model of the whole system (Universe plus instrument) that best reproduces the data. Whenever we have a model for the spectra of the sources (*e.g.* SN Ia's, galaxies of different type/redshift) we want to be able to analyze things this way.

Method B is preferred when we need to compare the data to external systems, or when we want to place all the data on the same system even though the instrument properties vary (*e.g.* for different airmasses, two slightly different QE curves or filters in same nominal band, etc.). It also has the advantage that the Analysis domain classes can just fit their source models to the standard-mag outputs, and don't ever have to examine the instrumental fluxes directly.

Figures 4 show the inputs and outputs of the calibration process in both Methods. In Method A, the calibration parameters (which specify the model passbands) are combined with a posited source spectrum to give a model instrumental output for each `Channel`. In Method B, the instrumental fluxes are *input*, along with the calibration parameters, and the output are estimated standard-band fluxes. Input includes uncertainties on the instrumental fluxes, which are propagated to a covariance matrix for the output fluxes.

I am reasonably sure that we will need to implement Method A. But astronomical calibration people are not accustomed to having to estimate passband functions, they usually just deal with moments of the passbands (like color terms, which are essentially 1st moments of the passband). There must be some way to bridge the gap between these two views of calibration, and come up with a uniform interface that can suit both. But I have not thought of it yet.

In any case we will need these kinds of classes:

```
class ParameterVector extends Noun;     //The parameters of the Calibration model
```

Table 1. SpectralUnits Definitions

Quantity	Units
Luminosity	$\gamma\ (\ln \lambda)^{-1}\ \mathrm{s}^{-1}$
Flux	$\gamma\ (\ln \lambda)^{-1}\ \mathrm{m}^{-2}\ \mathrm{s}^{-1}$
Area	m^2
SolidAngle	sr
SurfaceBrightness	$\gamma\ (\ln \lambda)^{-1}\ \mathrm{m}^{-2}\ \mathrm{s}^{-1}\ \mathrm{sr}^{-1}$
TransferFunction	(scalar)
Rate	s^{-1}
λ	μm

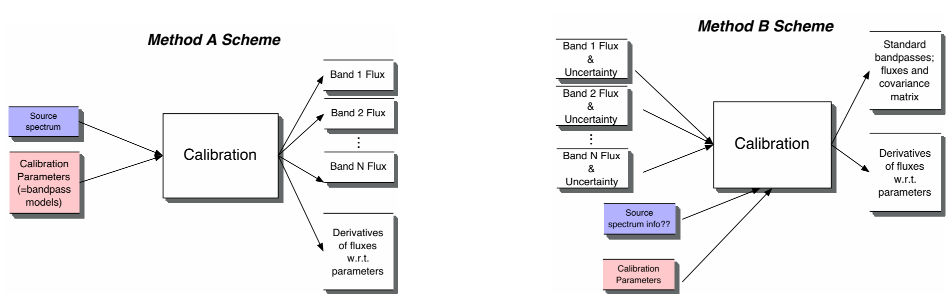

Fig. 4.— Two possible views of a Calibration process.

```
class ParameterCovariance extends Noun; //Uncertainties on the Calibration model
class CalibrationModel;    // ???? The A or B method black box itself
```

There are also classes that the calibrators themselves will use to refine the calibration model, using observations of standards, multiple observations of the same targets, etc.

The `ParameterVector, ParameterCovariance` classes will likely be defined in the Utility domain.

12. Analysis Domain

```
LightCurve
LightCurveFit
LightCurveObservables

Spectrum
SupernovaSpectrumFit
SupernovaSpectrumObservables

SupernovaFit //constrain SN mag with LC & spectrum
HubbleDiagram
HubbleDiagramFit
```